LLNL-CONF-653583

Applying Performance Modeling to Guide Hybrid MPI/OpenMP Use and Communication/Computation Tradeoff in Algebraic Multigrid

H. Gahvari, W. Gropp, K. Jordan, U. M. Yang

April 23, 2014

Supercomputing 2014
New Orleans, LA, United States
November 16, 2014 through November 21, 2014

Applying Performance Modeling to Guide Hybrid MPI/OpenMP Use and Communication/Computation Tradeoff in Algebraic Multigrid

Hormozd Gahvari*‡, William Gropp*, Kirk E. Jordan† and Ulrike Meier Yang‡

*Department of Computer Science, University of Illinois at Urbana Champaign, Urbana, IL 61801
†IBM TJ Watson Research Center, Cambridge, MA 02142
‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551

{gahvari,wgropp}@illinois.edu, kjordan@us.ibm.com, umyang@llnl.gov

Abstract—**Performance modeling is often employed to understand and project application performance. This paper takes it in another direction, applying performance modeling to provide practical direction when running the popular iterative solver algebraic multigrid (AMG). We make use of performance models to help users of AMG make two often difficult decisions regarding programming model and tradeoff of communication and computation. In the case of the former, we use a performance model for hybrid MPI/OpenMP to give users information on preferable on-node mixes of MPI tasks and OpenMP threads. For the latter, we apply a performance model at runtime to guide gathering and placement of data in communication-intensive portions of AMG onto a subset of the processes that trades communication for computation, improving performance on current-generation machines. We envision this kind of applied performance modeling to be of continued use for AMG and other applications in the future.**

I. Introduction

HPC applications today find themselves needing to adapt to a computing landscape that features more and more massive parallelism, expressed in many different ways in the underlying architecture. This has led to explorations into better tailoring the programming model to suit these new machines, which feature massive on-node parallelism, and better tailoring the applications themselves to run on massively parallel platforms.

The adaptations to both application and programming model bring with them a number of tunable parameters that need to be set properly in order for things to run well. One common programming model adjustment is to introduce the use of a threaded programming model like OpenMP within a message passing code written using MPI, so that some or even all of the on-node parallelism is handled through a more natural shared memory model. Getting the best performance then requires setting the correct mix of MPI tasks and OpenMP threads to use on the nodes of the machine.

A common application adjustment is to trade communication for computation, under the new conventional wisdom that computation is cheap but moving data is expensive. As true as this is, the extent of this tradeoff that is best to make requires a good amount of fine-tuning. Not enough of it brings too few of the benefits, and too much of it overwhelms the gains from the reduced communication with the added computation burden.

Performance modeling is often employed to better understand the inner workings of both applications and architectures, and this can help better inform application development and adaptation to the changing HPC landscape through in-depth performance analyses and projections. With all the information performance models are capable of providing, it is natural to want them to be able to guide the selection and adjustment of tunable parameters such as the ones described above, as there is potentially a lot of guesswork on the part of users as to how to best set them, and for users like scientists and engineers who are using HPC applications to perform simulations for their research, time that they would spend tuning parameters is time not spent directly conducting their research.

In this paper, we apply performance modeling to help set such tunable parameters for algebraic multigrid (AMG), a popular solver for large, sparse systems of linear equations that finds use in a wide range of scientific applications. We examine both hybrid MPI/OpenMP use on multicore nodes, and communication/computation tradeoff. In particular, we:

- Provide effective recommendations on the best mix of MPI tasks and OpenMP threads to use when running AMG on a machine with multicore nodes.
- Guide communication/computation tradeoff on the fly while running AMG, ensuring we perform an amount of it that improves performance.

The two benefits are cumulative, as the communication/computation tradeoff guided by the model can be used when running in a hybrid MPI/OpenMP configuration that is suggested by it earlier. Experiments on two modern HPC platforms validate our approach.

The remainder of this paper is organized as follows. Section II introduces AMG. Section III discusses the use of hybrid MPI/OpenMP programming in AMG, then introduces a performance model for hybrid MPI/OpenMP in AMG and adapts it to make suggestions for on-node mixes of MPI tasks and OpenMP threads. Section IV proceeds to adapt the performance model for runtime use, and then uses the model to guide a data gathering approach that achieves communication/computation tradeoff by combining the data sets of different processes. We give concluding remarks in Section V.

II. Algebraic Multigrid

Multigrid methods are efficient iterative solvers of many systems of sparse linear equations. When designed properly, they are algorithmically scalable, i.e. they can solve a linear system with N unknowns with only $\mathcal{O}(N)$ work. This property gives them the potential to solve ever larger problems on proportionally larger parallel machines in constant time. Multigrid methods achieve this optimality by employing two complementary processes: smoothing and coarse-grid correction. In the classical setting of scalar elliptic problems, the smoother (or relaxation method) is a simple iterative algorithm like Gauss-Seidel or weighted Jacobi that is effective at reducing high-frequency error. The remaining low-frequency error is then accurately represented and efficiently eliminated on coarser grids via the coarse-grid correction step. Algebraic multigrid (AMG) is a flexible and unique type of multigrid method that does not require geometric grid information. In AMG, coarse grids are simply subsets of the fine grid variables, and the coarsening and interpolation algorithms make use of the matrix entries to select variables and determine weights. These algorithms can be quite complex, particularly in parallel. While AMG can be used as a standalone solver, it is often used as a preconditioner for a Krylov method, such as conjugate gradient (CG) or GMRES. We will use it as a preconditioner for CG.

We will now define the AMG algorithm used in our experiments. Consider a linear system of the form $Au = b$, where A is sparse $n \times n$ matrix and u and b are vectors of size n. The components of AMG, i.e. interpolation, restriction and coarse grid operators, are determined in a first step, known as the *setup phase*:

1) Set $k = 0$, $A_0 = A$.
2) Determine the coarse grid variables using a coarsening algorithm.
3) Define the interpolation operator P^k_{k+1}.
4) Define the restriction operator R^k_{k+1} (here $R^k_{k+1} = (P^k)^T$).
5) Set $A_{k+1} = R^k_{k+1} A_k P^k$.
6) Set up a smoother S_k, if necessary.
7) If A_{k+1} is small enough, set $m = k + 1$ and stop. Otherwise, set $k = k + 1$ and go to step 2.

Once the setup phase is completed, the *solve phase*, can be performed. While there are various options on performing the solve phase, we are using a V-cycle, which is defined by the following algorithm, using an initial guess x_0:

```
$r_0 = b - A_0 x_0$
For $k = 1, ..., m-1$
    $x_k = 0$
end
For $k = 0, ..., m-1$
    $x_k = S_k^{-1}(r_k - A_k x_k)$
    $r_{k+1} = (P^k_{k+1})^T (r_k - A_k x_k)$
end
Solve $A_m x_m = r_m$.
For $k = m-1, ..., 0$
    $x_k := x_k + P^k_{k+1} x_{k+1}$
    $x_k := x_k + S_k^{-T}(r_k - A_k x_k)$
end
```

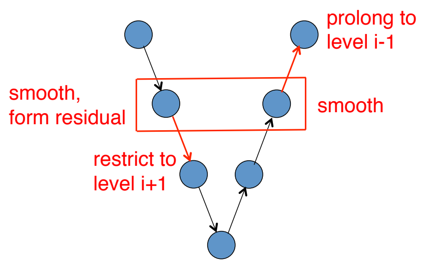


Fig. 1. AMG V-cycle with the fundamental operations at level l highlighted. These are smoothing, residual formation, restriction, and interpolation (also known as prolongation).

A summary of the operations at each level of an AMG V-cycle are in Figure 1.

In our experiments, we use the BoomerAMG solver [1] in the hypre software library [2]. On the finest level, we use aggressive coarsening with multipass interpolation [3]. On the following levels, HMIS coarsening [4] with extended+i interpolation [5] truncated to at most 4 elements per row is used. Our smoother is hybrid Gauss-Seidel, which is comprised of Gauss-Seidel iteration between process boundaries and Jacobi iteration across process boundaries, leading to block diagonal matrices S_k, where the i-th diagonal block is the lower triangular part of the local portion of A_k that is owned by process i and p is the number of processes. In all our experiments AMG is used as a preconditioner for the conjugate gradient method. More detailed information on AMG may be found in [6], [7].

Sparse matrices in BoomerAMG are stored in a ParCSR matrix data structure. In the data structure, if there are p MPI processes, the matrix A is partitioned by rows into matrices A_k, with $k = 0, \ldots, p-1$. Each matrix A_k is stored with a process as two sequential compressed sparse row (CSR) sparse matrices, $A_k = D_k + O_k$. D_k contains the entries in A_k with column indices that point to rows stored on process k. O_k contains the remaining entries. Computing a matrix vector product (MatVec) Ax involves for each process k computing $A_k x = D_k x^D + O_k x^O$, where x^D is the portion of x stored locally on process k and x^O is the portion that is stored on other processes, requiring communication. Further detail is available in [8]. Hybrid MPI/OpenMP use is accomplished by using OpenMP threads within MPI tasks at the loop level in the form of `parallel for` constructs, which spawn a number of threads that simultaneously execute portions of the loop being parallelized.

III. Hybrid MPI/OpenMP Prediction for AMG

One of the challenges to getting good performance from AMG is performance degradation on coarse grids owing to increased communication. Processes have more communication partners that are farther away. There is much less computation to keep them busy while waiting for data from these many partners. This was analyzed in depth in [9].

Hybrid MPI/OpenMP programming is attractive in this setting for its potential to alleviate these issues. It was explored in depth for AMG in [10] and [11], and found to be of benefit on two machines with multicore nodes, an Opteron cluster and a Cray XT5. Later studies [12], [13] looked at the Cray XK6 and the IBM Blue Gene/Q, and developed a performance model for AMG when run using hybrid MPI/OpenMP that was able to explain the observed performance on the tested machines. While using hybrid MPI/OpenMP in fact brought benefits in the form of communication reduction, there were also drawbacks in the form of reduced computation performance that limited the amount of OpenMP that could be utilized. The specific balance of communication reduction and computation slowdown varied by machine, which caused the best on-node mix of MPI tasks and OpenMP threads to vary.

The studies cited above focused on the effect of architecture on hybrid MPI/OpenMP performance, but the specific problem being solved is certain to have an effect on hybrid performance as well. Different problems, and in fact different sizes of the same problem run on the same number of cores, are going to have different balances of communication and computation. The range of possibilities for on-node mixes of MPI and OpenMP is also expanding with simultaneous multithreading (SMT), found on machines such as Blue Gene/Q. The end result is that users face a lot of potentially costly guesswork as to which on-node mix of MPI and OpenMP performs the best for their problem/machine pair.

Here, we present a relatively straightforward and efficient way for making informed decisions about the best on-node mix of MPI and OpenMP for AMG on a particular problem/machine pair. It leverages the hybrid MPI/OpenMP performance model for the AMG solve cycle from [12] ito suggest to users appropriate on-node mixes of MPI tasks and OpenMP threads. Over the course of this section, we introduce and summarize the performance model we will be using, apply it to generate suggested MPI/OpenMP mixes, and compare the suggestions to actual results.

A. Performance Model

The hybrid MPI/OpenMP performance model from [12] was based on going through each individual level of an AMG V-cycle and counting the fundamental operations: smoothing, forming the residual, restriction, and interpolation. Each operation was treated as sparse matrix-vector multiplication using the appropriate operator. Matrix entry and communication counts, available from hypre's data structures, were combined with measured machine parameters to form an analytical expression for the total cycle time.

We will be discussing the model in more detail in Section IV-B, when we go into adapting it for use at runtime. Here, we summarize the portions of the model that specifically deal with hybrid MPI/OpenMP. In its baseline form, the model splits the cost of each MatVec into its communication and computation components. The specific amount of computation is based on the number of matrix nonzero entries per core, while the specific amount of communication is based on the number of messages sent and the amount of data sent in those messages. This assumes that only MPI tasks are used for parallelism.

When adding OpenMP to the mix, the communication counts are assumed to change based on whatever the new number of MPI tasks. Computation counts also do not change, as they are based on the number of matrix nonzero entries per core. What does change, however, is the computation time, which is penalized based on two scenarios. The first is limited memory bandwidth. With no definite partitioning of local memory like there is in the message passing case, threads can contend with each other when accessing memory that is shared by multiple cores. This is taken into account when using t threads by multiplying the time per floating-point operation by $\frac{b_1}{b_t}$, where b_i is the memory bandwidth per thread when using i threads. The second is migration of threads across cores by the operating system. This can happen when running a hybrid MPI/OpenMP program, and when such migration occurs across cores that are on different sockets, there can be a significant decline in on-node performance. This is penalized with the worst case in mind. If p_{node} is the number of processors on a node, and t is the number of threads, then we multiply the time per floating-point operation by $\max\left\{1, \frac{t}{p_{\text{node}}}\right\}$. The overall penalized compuation time is obtained by multiplying the time per floating-point operation by both penalties.

B. Making Predictions

The performance model summarized above was initally created to be descriptive, to explain observed hybrid MPI/OpenMP performance in AMG. There are substantial challenges faced when trying to make it predictive so that it can give suggestions for on-node MPI/OpenMP mixes to use. In general, a preexisting AMG hierarchy with operator statistics and communication counts is not going to be available. And even if one were available, changing the MPI/OpenMP mix requires adjusting the communication counts.

In getting around these constraints, we are inspired by an approach taken in an early study of the exascale potential of FFTs and multigrid [14] that was based on performance models of both applications. With no actual exascale machine or even a design for one avaiable, the approach was to instead allow the machine parameters to vary and determine regions in the parameter space within which exaflop/s performance was possible. Similarly, without an AMG hierarchy complete with communication counts and operator statistics, we instead allow the amount of communication and computation in the cycle to vary and determine regions in the space of the two components when using a certain degree of hybrid MPI/OpenMP provides improvement over using only MPI. Specifically, we assume that an AMG cycle when run all-MPI has a certain percentage of its time devoted to computation, with the rest of the time devoted to communication. With the help of the OpenMP penalties from the performance model, we can calculate for a given mix of OpenMP threads and MPI tasks how much communication reduction is necessary to show an improvement in performance.

For simplicity of calculation, assume a cycle takes 100 seconds. We split the cycle time T_{cycle} into its communication and computation compoenents:

$$T_{\text{cycle}} = 100 = T_{\text{comp}} + T_{\text{comm}}$$

Define f_{comm} to be the fraction of the communication time needed to get an improvement using OpenMP. Given the penalty p_{omp} to the computation time for using OpenMP, we

TABLE I. MEMORY BANDWIDTH PER THREAD (MB/S) FOR VULCAN AND TITAN.

OpenMP Threads	Vulcan	Titan
1	3921.9	9919.6
2	3910.3	5107.3
4	3886.4	3666.0
8	3512.5	1909.1
16	1747.6	1392.1
32	881.00	–
64	434.44	–

want the region where

$$p_{\text{omp}} T_{\text{comp}} + f_{\text{comm}} T_{\text{comm}} \leq 100,$$

which we visualize by plotting the region

$$0 \leq f_{\text{comm}} \leq \frac{100 - p_{\text{omp}} T_{\text{comp}}}{100 - T_{\text{comp}}}$$

for T_{comp} ranging from 0 to 100 (corresponding to 0% and 100% computation in the cycle, respectively). Plotting this "improvement region" for different MPI/OpenMP mixes on the same axes gives a concrete idea of preferable MPI/OpenMP mixes for different problems on a particular machine.

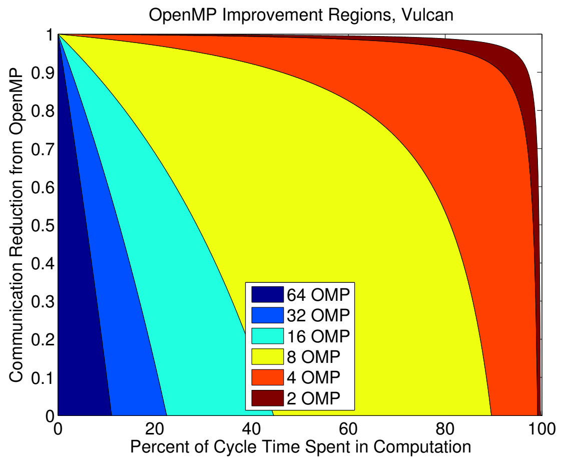
We are still left with predicting how much simultaneous multithreading to use. A performance model for simultaneous multithreading was developed for Blue Gene/Q [13] that treats it as an additional penalty to the computation rate. However, there is no generic multi-machine model for it. We currently assume full SMT use on the machine we test that has this capability, and leave determination of the amount of it to use as future work.

C. Prediction Results

We tested our prediction scheme on four different machines, using the 3D 7-point Laplacian as our model problem. To measure memory bandwidths for the OpenMP penalties, we used the STREAM benchmark [15]. For each machine, we plotted the corresponding hybrid MPI/OpenMP improvement region, and then compared the suggestions from the plots with actual results from running 7-point Laplace problems of varying sizes using varying on-node MPI/OpenMP mixes.

1) Machines: **Vulcan** is a 24,576 node IBM Blue Gene/Q at Lawrence Livermore National Laboratory. There is one 1.6 GHz 16 core processor per node, with SMT capability for up to 4 threads per core. The hardware bandwidth between nodes is 40 GB/s. All experiments use IBM's compiler, and the MPI implementation is an IBM-derived version of MPICH2.

Titan is an 18,688 node Cray XK7 at Oak Ridge National Laboratory. Each node features one 2.2 GHz 16 core AMD Opteron 6274 processor, which we treat as two 8 core processors for the purposes of the hybrid model because the AMD Opteron 6200 series processor actually consists of two dies with eight cores per die [16]. Each node also features one NVIDIA Tesla K20 GPU, which we do not consider here. The nodes are connected by Cray's Gemini interconnect, which features a 3D torus topology with a hardware bandwidth of 20.8 GB/s between nodes. All experiments use the PGI compiler, version 13.10.0. The MPI implementation is Cray's native MPI.



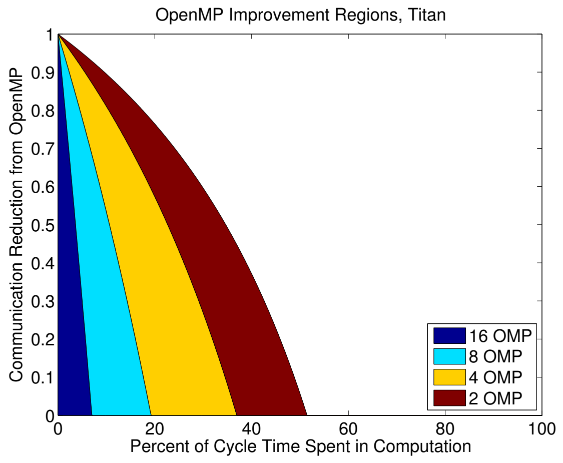


Fig. 2. Improvement regions for using hybrid MPI/OpenMP on Vulcan and Titan.

2) Results: Figure 2 shows the improvement regions for using a given number of OpenMP threads per MPI task on each machine, with the per-thread memory bandwidth numbers we used for the calculation in Table I. The results suggest that on Titan, only MPI should be used except for very communication-dominated problems. On Vulcan, the results lean towards using 4 or 8 OpenMP threads per MPI task, which correspond to 16 or 8 MPI tasks per node, respectively. Since Blue Gene machines have very good interconnects, it is likely that using 4 OpenMP threads in 16 MPI tasks is the best choice.

3) Comparison with Actual Results: To see how well we did with predicting the right mix of MPI/OpenMP mix, we used AMG to solve 3D 7-point Laplacians on 512 cores of each machine we evaluated, for problems ranging from $10 \times 10 \times 10$ points per core to $50 \times 50 \times 50$ points per core, running 20 AMG cycles. The average cycle times are in Figure 3. On Vulcan the best configurations overall were 8 MPI tasks per node and 16 MPI tasks per node. Greater numbers of MPI tasks per node became competitive only for the two largest problem sizes, and the best configurations were still fairly competitive. Overall, the results matched the improvement regions well.

On Titan, the best cycle time was for running MPI only in all cases, except for the smallest problem, for which the

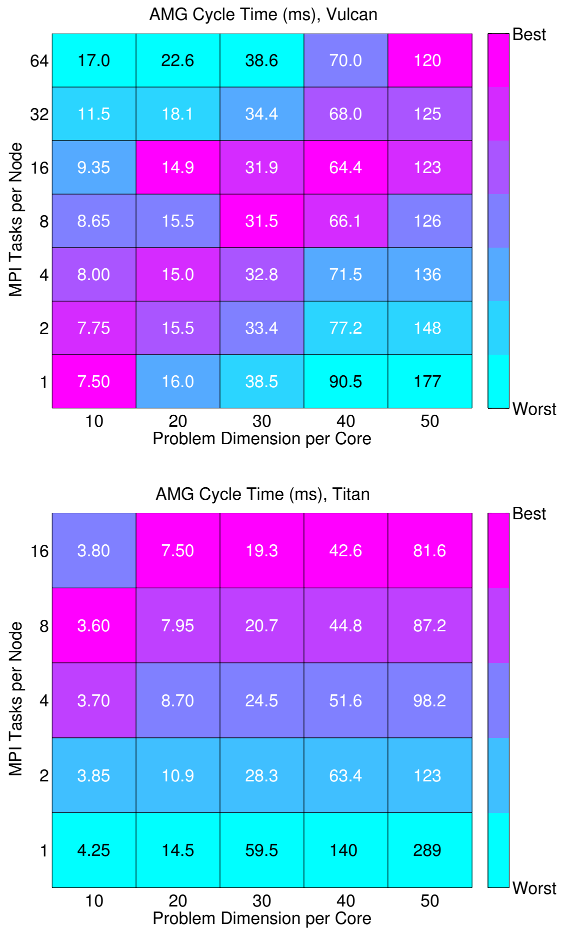


Fig. 3. Cycle times when using AMG to solve the 7-point Laplacian for a variety of problem sizes and MPI/OpenMP mixes on Vulcan and Titan.

best cycle time occurred running with 8 MPI tasks per node and 2 OpenMP threads per MPI task. This mostly matched the improvement regions, save for the inability of configurations with more OpenMP to be competitive for the smallest problem. The 8 MPI/2 OpenMP configuration was also a close second for the other problem sizes.

IV. Guiding Data Redistribution in AMG

Having explored using a performance model to guide the use of hybrid MPI/OpenMP for AMG, we now turn towards applying it at another phase of the algorithm. We use a performance model to guide data redistribution during the AMG setup phase and solve cycle that reduces communication at the expense of added computation.

Data redistribution to reduce communication has a rich history owing to potential gains from reducing communication. Gropp [17], using a very basic performance model, suggested that on one of the coarse grids, distributing the problem data redundantly across every involved processor would improve performance. This feature was later added to hypre starting with version 2.8.0b [18], and yielded substantial speedups for AMG in some cases [19], but the benefits were found to diminish at scale. Womble and Young [20] implemented a geometric multigrid method that had pairs of communicating processors with 2 or fewer unknowns in any dimension combine and replicate their data, and reported improved parallel efficiency.

Others redistributed data without using redundancy, concentrating data onto fewer processes on coarse grids to reduce communication. Nakajima [21] concentrated data onto one process on the coarsest grid in a geometric multigrid solver and then further deepened the multigrid cycle on that process. The smoothed aggregation AMG solver ML from Sandia National Laboratories [22] allows for the concentration of data on coarse grids onto fewer processes for load balancing and to prevent convergence degradation. Sampath and Biros [23] also concentrated data for load balance purposes in an octree-based geometric multigrid solver.

These methods face two big challenges. The first is deciding at what level in the multigrid hierarchy to perform redistribution. Performing it at too fine of a level causes an overload of computation which wipes out any benefit from reducing communication. Performing it at too coarse of a level results in marginal gains at best, due to missing opportunities to reduce communication in costlier parts of the cycle. The second is the degree of redistribution. The amount of problem data concentrated onto the individual processes in a redistribution scheme can be varied, which changes the amount of communication reduction and computation increase. When redistribution brings benefit, the best choice is usually somewhere in between eliminating communication entirely and leaving things unchanged.

The methods mentioned above either have fixed redistribution criteria or control it through parameters that potentially need a lot of guesswork to determine correctly. hypre currently has the level at which redundancy is employed as an option to be set by the user, in the form of a number of unknows below which the switch is made. ML has that plus a few other user-specifiable options to control data redistribution. Sampath and Biros used a minimum grain size, below which redistribution was performed to ensure load balance. Their minimum grain size was determined using a heuristic, but in follow-up work [24], Sundar et. al. noted that it may need to be determined empirically.

This is where performance modeling comes in. We tie data redistribution to a performance model that we can employ at runtime to decide both when to perform data gathering, and how much of it should be performed, to ensure that we get a benefit versus doing nothing at all. Over the remainder of this section, we describe our approach, show how we tie it to a performance model, and demonstrate speedups when using it in practice on modern parallel machines.

A. *Redistribution Approach*

Our data redistribution approach follows the work of Gahvari, et. al. [25], which divided the problem domain into chunks and assigned processes to them. Each process owned everything in its chunk, making the data redistribution a redundant one. A mapping strategy was used in conjunction with a cyclic mapping of MPI ranks to nodes to confine computation to nodes, and a performance model was used to make the redistribution decision. Speedups of up to 2-3x were observed on a multicore Linux cluster with very poor network performance running on up to 4096 cores, and the programming model was MPI only.

We diverge slightly from their approach, however, due to a couple of considerations. First, we employ nonredundant data gathering. The reason is that we want to run on larger processor counts, and there is a risk of running into either a memory-based or MPI implementation-based upper limit on the number of new MPI communicators we can create [26]. We also do not consider localization of communication to nodes here. While such localization has much potential for performance improvement, we are here examining machines with much more modern interconnects; without a good model of how things change when going to only on-node communication, which we do not have yet, we would rather tread with caution and avoid an incorrect decision as to when to perform data redistribution and how much of it to perform.

What we do is, on each grid during the AMG setup phase except for the finest one, make a decision to coarsen and form the next grid or redistribute first before resuming coarsening. This decision is made with the help of a performance model, which estimates the cost of redistributing and using the redistributed operators versus not doing so, and makes a decision based on these cost estimates. The performance model depends on measurable machine parameters and information that is readily available from the data structure in hypre. The redistribution itself involves groups of MPI ranks that still have data combining it through `MPI_Gatherv` operations, and each group storing its pooled data on one of its members. For simplicity, we divide the participating MPI ranks (the ones that have data) into a number of chunks. If this number does not divide the ranks evenly, then, if P is the number of processes and C is the number of chunks, the first $P \bmod C$ chunks have $\left\lceil \frac{P}{C} \right\rceil$ processes, and the rest have $\left\lfloor \frac{P}{C} \right\rfloor$ processes.

After redistribution, setup proceeds on the redistributed operator over the processes that still have data. Each such process will have only $C-1$ neighbors at most to communicate with, but more computation to do as well than if there were no redistribution, hence the communication/computation tradeoff. We assume here that C is less than the maximum over all processes of the number of messages being sent, to ensure that we reduce communication. The solve cycle will, beyond the point of redistribution, also use the redistributed operator, with involved processes having to perform the same `MPI_Gatherv` operations to combine their components of the solution vector. When progressing from coarse to fine, passing the redistribution point will involve the root process of each `MPI_Gatherv` performing an `MPI_Scatterv` operation to send the appropriate parts of the solution vector to the processes that need them.

We next explain how we use the performance model at runtime, and then show results of using it to guide data redistribution on the same machines where we examined selecting the mix of MPI tasks and OpenMP threads.

B. Guiding with a Performance Model

To guide data redistribution, we use the same performance model for the solve cycle that served us previously when we examined selecting the mix of MPI tasks and OpenMP threads to use. As we will explain here, it can also be readily adapted to guiding the redistribution when using hybrid MPI/OpenMP, an expansion on the prior MPI only work of Gahvari, et. al. But first, we will need to explain more of its details.

1) Model Details: We begin counting matrix-vector multiplications. At each level of an AMG cycle, there are five matrix-vector multiplications or equivalent operations. Three involve the solve operator: the two smoothing steps and residual formation. Two involve the interpolation operator: restriction and interpolation. The interpolation operator, however, is not available to us because forming it would require coarsening to form the next coarsest grid, a step we cannot undertake before deciding whether or not we are going to redistribute data first. We instead approximate the MatVecs that require the interpolation operator with MatVecs that use the solve operator.

Our model for a MatVec is as follows. We express it first in terms of a baseline α-β model, which gives each message sent a cost of $\alpha + n\beta$, where α is the communication start-up time, and β is the cost to send one double-precision floating-point value, and then add penalties to take architecture into account as was done in [9] and [12]. We first define the following parameters in addition to α and β:

- N_i – number of nonzero entries in the level i solve operator
- P_i – number of processes with data on level i
- p_i – maximum number of sends in the level i solve operator over all involved processes
- n_i – maximum number of elements sent in the level i solve operator over all involved processes
- t_i – time per floating-point operation on level i

The time to perform a MatVec with the level i solve operator then becomes

$$T^i_{\text{matvec}}(\alpha, \beta) = 2\frac{N_i}{P_i}t_i + p_i\alpha + n_i\beta.$$

We augment this baseline model with additional terms and penalties to reflect issues seen on real machines. These penalties can be "on" or "off" depending on the machine, and the best fit for a particular machine will have some penalties on and some penalties off. Our first penalty is to add a γ term that represents the delay per hop when sending a message, to take into account messages traveling long distances. The corresponding change in the baseline model is to replace α with

$$\alpha(h) = \alpha(h_m) + (h - h_m)\gamma,$$

where h is the number of hops a message travels, and h_m is the smallest possible number of hops a message can travel in the network. We assume h to be the diameter of the network within the job's partition to take routing delays into account. h_m is 1 in a torus or mesh network, and 2 in fat-tree networks, in which a message travels through at least one switch, which involves using two links.

Another issue seen in practice is limited bandwidth. Message passing applications have difficulty achieving the full hardware-provided bandwidth in ideal conditions. The bandwidth they can achieve in ideal conditions is in turn difficult to achieve in non-ideal conditions. Limited bandwidth also arises due to contention from messages sharing links. We take both of these into account with a penalty to β. Let $B_{\max}$ be the peak aggregate per-node bandwidth, and let B be the measured bandwidth corresponding to β, which is $\frac{8}{\beta}$ if β is the time to send one double-precision floating-point value. The penalty for being unable to achieve the full hardware bandwidth is $\frac{B_{\max}}{B}$. To account for link contention, we let m be the number of messages in the network, and l be the number of links available to the job. The link contention portion of the penalty is then $\frac{m}{l}$. The overall penalty to β is multiplication by the sum of both of these terms: $\beta \leftarrow \left(\frac{B_{\max}}{B} + \frac{m}{l}\right)\beta$.

Two other penalties stem from multicore nodes, which bring the possibility of increased contention between cores when accessing the interconnect, and increased contention in switches caused by the extra messages coming from the cores. We model this by multiplying $\alpha(h_m)$ and γ by $\left\lceil c\frac{P_i}{P}\right\rceil$, where c is the number of cores per node, and P is the total number of MPI ranks, both with data and without data. The penalized terms are $\alpha \leftarrow \left\lceil c\frac{P_i}{P}\right\rceil \alpha(h_m)$ and $\gamma \leftarrow \left\lceil c\frac{P_i}{P}\right\rceil \gamma$.

One last set of penalties stems from using hybrid MPI/OpenMP; however these were already covered in Section III-A.

2) Making the Redistribution Decision: When making the redistribution decision, at each level l, we use the performance model to compare two times: T^l_{noswitch}, the time spent on level l of the solve cycle if we do no redistribution, and T^l_{switch}, the time spent on level l if we do data redistribution and then use the redistributed operator in the solve cycle, and switch at the finest level at which $T^l_{\text{switch}} < T^l_{\text{noswitch}}$. We test this for different numbers of chunks of processes, starting from the smallest power of two less than p_l and searching over that and all smaller powers of two until we reach 1, which is the absolute minimum (the fully gathered case, which has no communication).

Assuming correct values for both, it is in fact, as Gahvari, et. al., noted [25] best to switch at this level. Assume there are L levels in the multigrid hierarchy, numbered 0 to $L-1$. If we switch at a finer level $\hat{l} < l$, then the decision we made will have been suboptimal due to a slowdown at levels $\hat{l}$ through $l-1$ combined with the improvements at levels l through $L-1$ we would have obtained from the original switch. If we switch at a coarser level $\hat{l} > l$, we will again have made a suboptimal decision – switching at level l would give the improvements from switching at level $\hat{l}$ combined with improvements from switching at levels l through $\hat{l}-1$ that switching at level $\hat{l}$ would not have provided. We will not, however, have correct values for T_{switch} and T_{noswitch}, as they are based on approximations. To counteract this, we will introduce a pair of safeguards to prevent overeager switching decisions.

The first step in the decision is to determine T^l_{noswitch}. This is the more straightforward part. We use the time spent to perform 5 MatVecs using the level l solve operator to represent the two smoothing steps, residual formation, restriction, and interpolation. This gives us

$$T^l_{\text{noswitch}}(\alpha,\beta) = 10\frac{N_l}{P_l}t_l + 2(p_l\alpha + n_l\beta).$$

The network parameters can be obtained through measurements; we explain how we obtain them for our experiments in Section IV-C. The rest of the information can be obtained by querying hypre's data structures, except for the computation time t_l. This is allowed to vary in the model because the computation rate for sparse matrix-vector multiply has been observed to vary greatly depending on the size of the solve operator and pattern and number of nonzero entries in it [27]. With no *a priori* way to determine it, we instead measure it by performing 10 sequential sparse MatVecs using the locally stored sparse matrix D_k on each process and dividing the observed time by the number of flops performed. We exclude processes that have no data, and take the maximum reported value over all processes to be t_l. However, if this value is greater than the one for t_{l-1}, we set $t_l = t_{l-1}$ and set $t_k = t_{l-1}$ for all levels $k > l$. This occurs when processes are close to running out of data. They measure an abnormally high time per flop due to primarily measuring loop overhead instead. Progression from fine to coarse in AMG results in decreasing matrix dimension and increasing matrix density, conditions under which the time per flop decreases in practice [27]. We measure like this even when using hybrid MPI/OpenMP and simultaneous multithreading; in that case, the measured t_l and the expression for t_l in the model equations is assumed to implicitly contain the applicable penalties.

Determining T^l_{switch} is a more involved process. Assume there are C chunks of processes. There are two components. The first are the collective operations that redistribute the solution vector. We need to charge two gather operations (for the solution and right-hand size vectors) and a scatter operation (for the solution vector). Each operation is assumed to take place over a binary tree over $\frac{P_l}{C}$ processes, incurring $\left\lceil \log_2 \frac{P_l}{C}\right\rceil$ sends. In each gather operation, if there are C_l unknowns in the solve operator at level l, counting from the root, each stage of gathering data on the tree involves sends of approximately size $\frac{C_l}{2C}, \frac{C_l}{4C}, \frac{C_l}{8C}, \ldots$, which we charge as $\frac{C_l}{C}\left(\frac{1}{1-\frac{1}{2}} - 1\right) = \frac{C_l}{C}$ units of data sent. The scatter operation is assumed to send approximately $\frac{C_l}{C}\left\lceil \log_2 \frac{P_l}{C}\right\rceil$ units of data. Using the baseline model, we write the time spent in collective operations as

$$T^l_{\text{collective}}(\alpha,\beta) = 3\left\lceil \log_2 \frac{P_l}{C}\right\rceil \alpha + \frac{C_l}{C}\left(2 + \left\lceil \log_2 \frac{P_l}{C}\right\rceil\right)\beta.$$

The next step is to determine the cost of a MatVec using the redistributed operator. We assume equal division of the number of nonzero entries in the original matrix among the gathered chunks, and assume equal distribution of the amount of data sent per message among the number of communication partners in the nonredistributed operator. We further assume that each process participating in the redistributed operator has $C-1$ communication partners, which is the most communication partners a process could possibly have so long as there are fewer chunks than communication processors in the original operator, which we assume. The network parameters do not change, but the time per floating-point operation does. However, we have no way of measuring it, so we leave it the same and use the first of the two safeguards we referred to earlier to keep it from being too much higher. The MatVec cost is

$$T^l_{\text{new\_matvec}}(\alpha,\beta) = 2\frac{N_l}{C}t_l + (C-1)\left(\alpha + \frac{n_l}{p_l}\beta\right),$$

and the time we get for T^l_{switch} is

$$T^l_{\text{switch}}(\alpha, \beta) = 5T^l_{\text{new\_matvec}}(\alpha, \beta) + T^l_{\text{collective}}(\alpha, \beta).$$

Our safeguard against increasing t_l is to prevent the local storage for the redistributed operator on each participating process from increasing too much in size. To determine how much is too much, we first classify the local MatVec operations we performed when measuring t_l into one of three categories, which are explained in more detail in [27]. For categorization, the cache size is determined by dividing the size of the shared on-node cache by the number of MPI processes per node. This is done to take hybrid MPI/OpenMP into account.

- Small: the matrix and the source vector fit in cache
- Medium: the source vector fits in cache, but the matrix does not
- Large: the source vector does not fit in cache

The time per floating-point operation was found in [27] to undergo significant jumps when moving from a smaller category to a larger one. If data gathering causes this, it will have a dramatic effect on performance. Thus, when evaluating numbers of chunks to group processes into, we exclude values that result in the problem category being at least halfway towards one of the larger ones. The constraint is set at going halfway rather than the more generous constraint of going all the way because performance can degrade even well before a category boundary is crossed; the extent to which this happens depends on the problem and the machine.

One other safeguard we build into this process is not to switch if it is not expected to have a big impact on the overall cycle time. Specifically, we keep track of the predicted cycle time according to the model, keeping a running sum that is updated at each level of the setup phase. If there is a projected gain from switching, but that gain is projected to be at most 5%, then we do not switch. Given a choice, we would rather miss speedups than slow the cycle down further, and since our model is based on approximations, we do not want to risk a slowdowns chasing small gains. To put things another way, we have the maxim, "Do no harm," from the world of medicine in mind.

C. Results

We tested data redistribution guided by the performance model on Vulcan and Titan, the same machines we considered before, on the same 3D 7-point Laplacian from Section III-C, with a problem size of $30 \times 30 \times 30$ points per core. We ran the problem with redistribution both enabled and disabled, taking the average of 10 trials to report our measured timings. Network parameters were measured using the latency-bandwidth benchmark HPC Challenge benchmark suite [28], and are reported in Table II. The best measured latency was taken to be $\alpha(h_m)$ and the the best measured bandwidth to be β. γ was determined from the network topology, $\alpha(h_m)$, and the worst latency measured by the benchmark. Along with the γ term, the bandwidth penalties to the baseline performance model were used for both machines, as they were observed on a prior Blue Gene/Q [13] and on a Cray XK6, which has the same interconnect as the XK7 [12]. The multicore penalties to

TABLE II. NETWORK PARAMETERS FOR VULCAN AND TITAN.

Parameter	Vulcan	Titan
α	3.93 μs	1.80 μs
β	4.54 ns	1.31 ns
γ	88.3 ns	155 ns

α and γ were not observed to apply on these machines, so we do not apply them here.

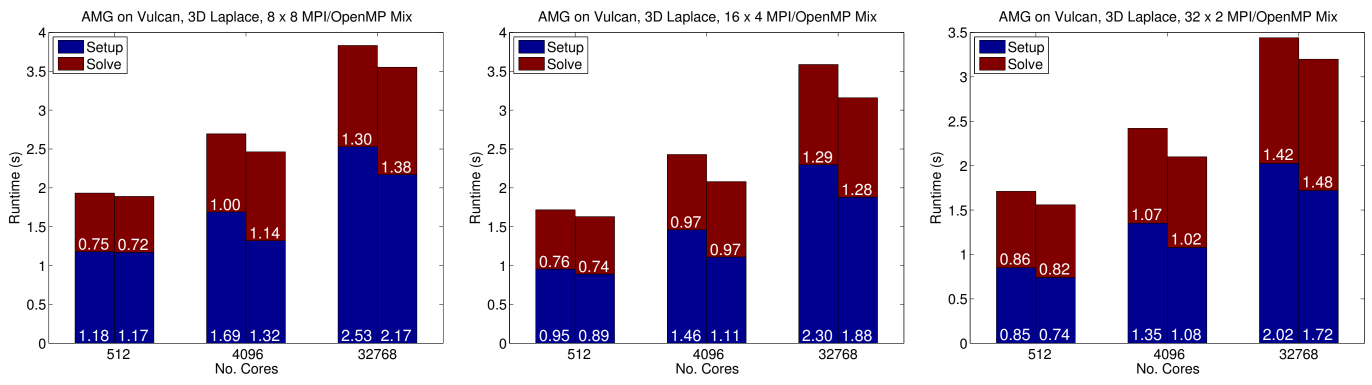
We used three different mixes of MPI and OpenMP per node on each machine, to show the ability of the model to guide redistribution under varying hybrid MPI/OpenMP settings. Runs were on 512, 4096, and 32768 cores, using the default block mapping of MPI ranks to nodes on the machine, which fills up each node with MPI ranks before moving onto the next one. Results for Vulcan are in Figure 4, and results for Titan are in Figure 5. The selected mixes were the top three from the prediction results in Section III-C.

Overall speedups ranged from 2% to 44%, and were distributed within that range. Large scale runs on Titan tended to yield the best speedups, and substantial performance degradation was seen in the setup phase for the mixes with more MPI tasks. This ran contrary to the earlier suggestion that running all MPI on Titan was the best choice, though the solve phase, which is what the model focuses on, was not overwhelmingly worse running all-MPI than when running with OpenMP. For the case of Vulcan, the overall best performer was 16 MPI tasks per node with 4 OpenMP threads per task, which the model had earlier suggested. Using 8 MPI tasks per node proved inferior to using 32 MPI tasks per node, which further hints at computation being the big contributor to the overall cost on Blue Gene machines.

V. Conclusions

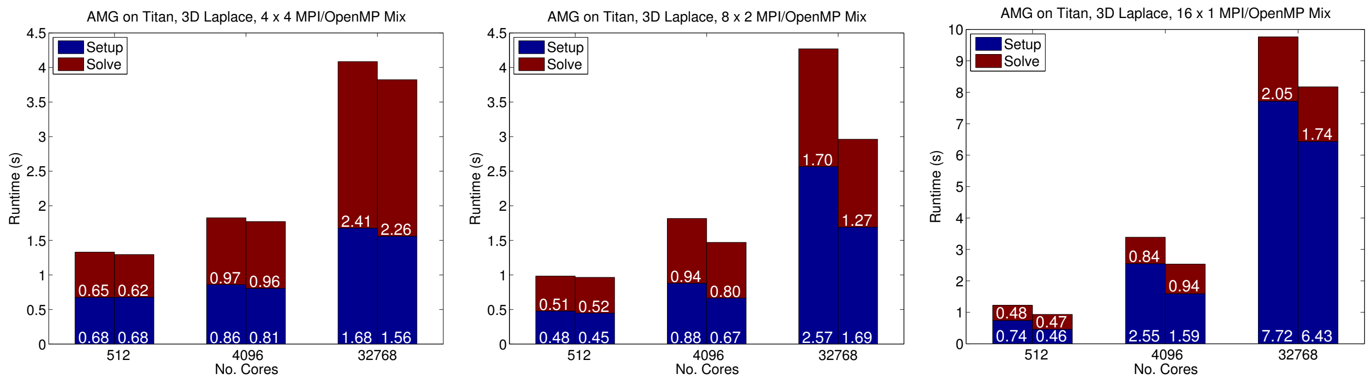
In this paper, we were successfully able to adapt a previously descriptive performance model for AMG when run using hybrid MPI/OpenMP so that we could make predictive use of it in two areas. The first was to guide the choice of hybrid MPI/OpenMP when running AMG, which we did by suggesting preferable on-node mixes of it based on the model. The suggested mixes were for the most part validated by actual results. Even when a suggested mix was not ideal, the model did still help by avoiding the selection of a mix that was going to certainly perform poorly. The second was to guide data redistribution during the multigrid cycle meant to trade communication for computation so that it would assuredly improve performance. Previous work had successfully used a performance model to guide redistribution when using only MPI, and we extended the use of model-guided redistribution to the hybrid MPI/OpenMP case and demonstrated its use on two modern parallel platforms at large scales.

Our results open a number of avenues for further study. With AMG specifically, we would like to be able to make more precise predictions on mixes of MPI tasks and OpenMP threads to use for running certian problems, as well as be able to make effective projections for large problems on future machines. We would also like to further refine our on-the-fly data redistribution, exploring how we can gain more from it and effectively introduce other additions like data localization that can be guided by the model. Another important task



	Speedups from Redistribution, 8 × 8 Mix			Speedups from Redistribution, 16 × 4 Mix			Speedups from Redistribution, 32 × 2 Mix		
	512 Cores	4096 Cores	32768 Cores	512 Cores	4096 Cores	32768 Cores	512 Cores	4096 Cores	32768 Cores
Setup	1.01	1.28	1.16	1.03	1.32	1.22	1.15	1.25	1.17
Solve	1.04	0.88	0.94	1.07	1.00	1.01	1.05	1.05	0.96
Overall	1.02	1.09	1.08	1.05	1.17	1.14	1.10	1.15	1.08

Fig. 4. Results and corresponding speedups for model-guided data redistribution for the 3D 7-point Laplace problem on Vulcan for varing mixes of MPI tasks and OpenMP threads per node. The bars on the left in each graph show timings when doing no redistribution, while the bars on the right show timings when doing redistribution.



	Speedups from Redistribution, 4 × 4 Mix			Speedups from Redistribution, 8 × 2 Mix			Speedups from Redistribution, 16 × 1 Mix		
	512 Cores	4096 Cores	32768 Cores	512 Cores	4096 Cores	32768 Cores	512 Cores	4096 Cores	32768 Cores
Setup	1.00	1.06	1.07	1.07	1.18	1.52	1.61	1.60	1.20
Solve	1.05	1.01	1.08	0.98	1.31	1.34	1.02	0.89	1.18
Overall	1.02	1.03	1.07	1.02	1.24	1.44	1.31	1.34	1.20

Fig. 5. Results and corresponding speedups for model-guided data redistribution for the 3D 7-point Laplace problem on Titan for varing mixes of MPI tasks and OpenMP threads per node. The 16 × 1 mix represents all MPI and no OpenMP. The bars on the left in each graph show timings when doing no redistribution, while the bars on the right show timings when doing redistribution.

for us is to model the setup phase. This phase was the one that benefited most from data redistribution. It was also what defied the hybrid MPI/OpenMP prediction. Having a model of it would allow us to greatly improve our data redistribution decisions and hybrid MPI/OpenMP suggestions.

There is also much room for further study not tied to our specific application. Successfully predicting how much to use a hybrid programming model when running applications in general, let alone AMG, is going to be a big question faced by users of large parallel machines, especially given the trend of increasing on-node parallelism. The number of possible choices for expressing this parallelism is going to expand accordingly, leaving users confronted by a lot of potential guesswork. Communication/computation tradeoffs are also going to be an increasing part of applications with increasing core counts in machines. As our model for AMG is based on modeling sparse matrix-vector multiplication, one direct avenue for expansion is to adapt it to other scientific applications that are based on it, and this is one area we plan on investigating.

Looking towards the future, we expect that, as architectures continue to evolve, there will be many more "knobs" that users can turn, in both the areas we have covered here, hybrid programming models and communication/computation tradeoff, and in other areas that have not yet been foreseen. Performance modeling, as we have demonstrated here, shows much promise in being able to turn these knobs for the users. This will allow them to focus on their research and more efficiently use their limited allocations on HPC machines to that end.

Acknowledgements

We thank Martin Schulz for valuable direction and input when this paper was in its early stages. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy awards DE-SC0004131 and DE-FG02-13ER26138/DE-SC0010049, and performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. It used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Neither Contractor, DOE, or the U.S. Government, nor any person action on their behalf: (a) makes any warranty or representation, express or implied, with respect to the information contained in this document; or (b) assumes any liabilities with respect to the use of, or damages resulting from the use of any information contained in this document.

References

[1] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, April 2002.

[2] "*hypre*: High performance preconditioners," http://www.llnl.gov/CASC/hypre/.

[3] U. M. Yang, "On long-range interpolation operators for aggressive coarsening," *Numerical Linear Algebra With Applications*, vol. 17, pp. 453–472, April 2010.

[4] H. De Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, pp. 1019–1039, 2006.

[5] H. De Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, "Distance-two interpolation for parallel algebraic multigrid," *Numerical Linear Algebra With Applications*, vol. 15, pp. 115–139, April 2008.

[6] A. Brandt, S. McCormick, and J. Ruge, "Algebraic multigrid (AMG) for automatic multigrid solutions with application to geodetic computations," Inst. for Computational Studies, Fort Collins, Colorado, Tech. Rep., 1982.

[7] K. Stüben, "An introduction to algebraic multigrid," in *Multigrid*, U. Trottenberg, C. Oosterlee, and A. Schüller, Eds. San Diego, CA: Academic Press, 2001, pp. 413–528.

[8] R. D. Falgout, J. E. Jones, and U. M. Yang, "Pursuing Scalability for *hypre*'s Conceptual Interfaces," *ACM Transactions on Mathematical Software*, vol. 31, pp. 326–350, September 2005.

[9] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms," in *25th ACM International Conference on Supercomputing*, Tucson, AZ, June 2011.

[10] A. H. Baker, M. Schulz, and U. M. Yang, "On the Performance of an Algebraic Multigrid Solver on Multicore Clusters," in *VECPAR'10: 9th International Meeting on High Performance Computing for Computational Science*, Berkeley, CA, June 2010.

[11] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of Scaling Algebraic Multigrid across Modern Multicore Architectures," in *25th IEEE Parallel and Distributed Processing Symposium*, Anchorage, AK, May 2011.

[12] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms Using Hybrid MPI/OpenMP," in *41st International Conference on Parallel Processing*, Pittsburgh, PA, September 2012.

[13] ——, "Performance Modeling of Algebraic Multigrid on Blue Gene/Q: Lessons Learned," in *3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, Salt Lake City, UT, November 2012.

[14] H. Gahvari and W. Gropp, "An Introductory Exascale Feasibility Study for FFTs and Multigrid," in *24th IEEE International Parallel and Distributed Processing Symposium*, Atlanta, GA, April 2010.

[15] J. D. McCalpin, "Sustainable Memory Bandwidth in Current High Performance Computers," Advanced Systems Division, Silicon Graphics, Inc., Tech. Rep., 1995.

[16] "AMD Opteron 6200 Series Processors Linux Tuning Guide," http://developer.amd.com/resources/documentation-articles/developer-guides-manuals.

[17] W. Gropp, "Parallel Computing and Domain Decomposition," in *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations*, T. Chan, D. Keyes, G. Meurant, J. Scroggs, and R. Voigt, Eds. SIAM, 1992, pp. 349–361.

[18] "*hypre* Reference Manual, Version 2.8.0b," https://computation.llnl.gov/casc/hypre/download/hypre-2.8.0b_ref_manual.pdf.

[19] A. H. Baker, R. D. Falgout, H. Gahvari, T. Gamblin, W. Gropp, K. E. Jordan, T. V. Kolev, M. Schulz, and U. M. Yang, "Preparing Algebraic Multigrid for Exascale," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-533076, March 2012.

[20] D. E. Womble and B. C. Young, "A model and implementation of multigrid for massively parallel computers," *International Journal of High Speed Computing*, vol. 2, pp. 239–255, 1990.

[21] K. Nakajima, "New Strategy for Coarse Grid Solvers in Parallel Multigrid Methods using OpenMP/MPI Hybrid Programming Models," in *2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, New Orleans, LA, February 2012, pp. 93–102.

[22] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala, "ML 5.0 Smoothed Aggregation User's Guide," Sandia National Laboratories, Tech. Rep. SAND2006-2649, February 2007.

[23] R. S. Sampath and G. Biros, "A parallel geometric multigrid method for finite elements on octree meshes," *SIAM Journal on Scientific Computing*, vol. 32, pp. 1361–1392, 2010.

[24] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler, "Parallel Geometric-Algebraic Multigrid on Unstructured Forests of Octrees," in *SC12: Proceedings of the 2012 ACM/IEEE Conference on Supercomputing*, Salt Lake City, UT, November 2012.

[25] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Systematic Reduction of Data Movement in Algebraic Multigrid Solvers," in *5th Workshop on Large-Scale Parallel Processing*, Cambridge, MA, May 2013.

[26] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on a Million Processors," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer, 2009, pp. 20–30.

[27] H. Gahvari, "Benchmarking Sparse Matrix-Vector Multiply," Master's thesis, University of California, Berkeley, December 2006.

[28] J. Dongarra and P. Luszczek, "Introduction to the HPCChallenge Benchmark Suite," University of Tennessee, Knoxville, Tech. Rep. ICL-UT-05-01, March 2005.